Assignment 6 – Surfin' Report

Arushi Tyagi

CSE 13S – Winter 24

Purpose

Audience for this section: Pretend that you are working in industry, and write this paragraph for your boss. You are answering the basic question, "What does this thing do?". This section can be short. A single paragraph is okay.
The purpose of this assignment is to traverse through graphs and find the most optimal path. We will be working with graphs and paths and taking in a file which contains the graphs. Going through multiple pathways, we will use depth-first search to find the shortest path and the Hamiltonian cycle.

Questions

Please answer the following questions before you start coding. They will help guide you through the assignment. To make the grader's life easier, please do not remove the questions, and simply put your answers below the text of each question.

- What benefits do adjacency lists have? What about adjacency matrices?
  Adjacency lists help us figure out how where edges start and end and what the weight is. Adjacency matrices help us figure out the neighbors of each node and the weight between them as well. They specifically make it easier however to see what nodes are next to each other.

- Which one will you use. Why did we choose that
  We will use both. The graphs text files use the adjacency lists because they give the information of each edge and how much each edge weighs. Then we will use the adjacency matrix to store the weights in graph.c. This also takes less space.

- If we have found a valid path, do we have to keep looking? Why or why not?
  Yes because there may be a shorter and more optimal path if we keep traversing through the multiple different options.

- If we find 2 paths with the same weights, which one do we choose?
  We will choose the one that we find first.

- Is the path that is chosen deterministic? Why or why not?
  It is deterministic because the neighbors of each node stays the same. The outcome of each path's traversal is the same.

- What type of graph does this assignment use? Describe it as best as you can
  It uses a directed weighted graph because the file inputs contain the starting point, ending point, and the weights of each edge. The weights of the graphs are also different if the starting and ending nodes are switched.

- What constraints do the edge weights have (think about this one in context of Alissa)? How could we optimize our dfs further using some of the constraints we have
  The weights represent how long it takes for Alissa to get from point A to point B or the distance between the two points. The cycle would have to contain the least weight between all the visited nodes

and they can only be visited once. To optimize the dfs, we could start from a point and go to another location that has the least weight at that edge and then traverse through the nodes. Also, while looking through all possible paths, she skips the nodes that have already been visited.

Testing

List what you will do to test your code. Make sure this is comprehensive. [1] Be sure to test inputs with delays and a wide range of files/characters.

- File with no path found
- Large file with a large number of vertices and a possible Hamiltonian cycle
- Small file with a small number of vertices and a possible Hamiltonian cycle
- Loading an invalid graph
- Wrong number of arguments in the input

How to Use the Program

Audience: Write this section for the user of your program. You are answering the basic question, "How do I use this thing?". Don't copy the assignment exactly; explain this in your own words. This section will be longer for a more complicated program and shorter for a less complicated program. You should show how to compile and run your program. You should also describe any optional flags or inputs that your program uses, and what they do.
There is a folder called maps which contains multiple graphs. In order to compile the program the user needs to enter "make" into the terminal. To run the program, the user has to type in ./tsp and then -d for a directed graph (the program automatically assumes an undirected graph), -i to read from an input file, -o to write to an output file, and -h to print a help message. These arguments will be followed by the graph file in the form of maps/.¡graph name¿

Program Design

Audience: Write this section for someone who will maintain your program. In industry you maintain your own programs, and so your audience could be future you! List the main data structures and the main algorithms. You are answering the basic question, "How is this thing organized so that I can have a chance of fixing it?". This section will be longer for a more complicated program and shorter for a less complicated program.
There are four main files. The first is stack.c which contains the functions that a stack performs including pop, push, free, peek, empty, copy, and print. This stack structure is used in path.c which contains functions to manage each path traversed in the graph. Stack and path structures are both created and there are functions to free the path and manage it. When traversing through the paths, the program adds a vertex to the path and recalculates the total weight. Similarly, functions to destroy and remove the paths are implemented as well. In graph.c, a graph structure is created with the number of vertices, a directed boolean, an array of visited booleans, an array of pointers to the names of the nodes, and an array of pointers to the weight of each edge. The graph file contains multiple functions similar to path.c and mostly deals with using the input from the graph files given in the input. It gets the names of the vertices, the number of vertices, the number of edges, and the weights. It also returns all these values which are then used in tsp.c. tsp.c contains the main function which reads the input and checks if it is a directed, undirected, weighted, or unweighted graph. It also performs dfs to find the shortest Hamiltonian cycle.

[1] This question is a whole lot more vague than it has been the last few assignments. Continue to answer it with the same level of detail and thought.

Pseudocode

Give the reader a top down description of your code! How will you break it down? What features will your code have? How will you implement each function.
stack.c: creates and manages stacks used for path.c

- stack_push. makes the value equal to the top value in the item stack. increment top.
- stack_pop. checks if stack is empty. if not, decrement top value of stack. value = s− >items[s− >top]
- stack_peek. checks if stack is empty. if not, val = s− >items[s− >top-1]
- stack_empty. makes the top of the stack equal to 0.
- stack_full. s− >capacity == s− >top
- stack_size. return s− >top
- stack_copy. loops through the whole stack. copies the items from the source to destination array
- stack_print. loops through the whole stack. prints each city by indexing through each item.

path.c: contains a path data structure and implements its different functions

- path_create. allocate memory for the path. update the vertices into the stack.
- path_free. Frees the path by freeing the value assigned to the vertex pointer.
- path_vertices. return stack_size(p− >vertices);
- path_distance. return p− >total_weight;
- path_add. peek the vertex to the top value of the stack and add the edge weight to the total weight.
- path_remove. pop the vertex from the stack and decrease the total weight by the edge.
- path_copy. copy the vertices from source to the destination's vertices.
- path_print. print the names according to the vertices

graph.c

- graph_create. Create the graph by allocating memory for the visited nodes, vertices, if the graph is directed, names of the nodes, and the weights.
- graph_free. Free the graph names, visited values, and weights. (ex: free((*gp)− >names)
- graph_vertices. Return the graph vertices. g− >vertices
- graph_add_vertex. free the name at the pointer of the vertex. g− >names[v] = strdup(name)
- graph_get_vertex_name. return g− >names[v]
- graph_add_edge. calculate weight depending on whether it is directed or undirected. If directed: g− >weights[start][end] = weight; If undirected: g− >weights[end][start] = weight;
- graph_get_weight. return g− >weights[start][end]
- graph_visit_vertex. g− >visited[v] = true;
- graph_unvisit_vertex. g− >visited[v] = false;
- graph_unvisited. return g− >visited[v];

- graph_print. loop through all the start and end vertices and print the weights for each value in the adjacency matrix.
- graph_destroy. free the graph

tsp.c

- read_file: takes in the file and checks if the graph is directed or undirected. Uses scanf and fgets to read the node and edge. Goes through each line and scans the inputs. As it reads the vertices, it adds them to the graph. As it goes through the edges, it also adds them to the graph.
- dfs: Goes through a vertex in the path and gets the edge weight by calling graph_get_weight. Then calls dfs until it gets to the end of the path. Once it has gone through all vertices in the graph, it checks if it is the smallest path. It keeps going until it has gone through all the paths and has found the shortest path.
- main: takes in the file and checks if it is a valid input. If it is 'i' then print to stderr "invalid input file". If it is 'o' print to stderr "invalid output file". If it is 'd' then it means the directed boolean is true. If it is 'h' then print the help message.

Function Descriptions

For each function in your program, you will need to explain your thought process. This means doing the following Do not simply use your code to describe this. This section should be readable to a person with little to no code knowledge. **DO NOT JUST PUT THE FUNCTION SIGNATURES HERE. MORE EXPLANATION IS REQUIRED.**

stack.c

- Function: stack_push. Inputs: stack, value. Outputs: boolean (true, false). Purpose: pushes a value onto the stack and returns a boolean that indicates the success of it.
- Function: stack_pop. Inputs: stack, value. Outputs: boolean (true, false). Purpose: pops a value out of the stack and returns a boolean that indicates the success of it.
- Function: stack_peek. Inputs: stack, value. Outputs: boolean (true, false). Purpose: Sets a value as the top of the items list and returns a boolean that indicates the success of it.
- Function: stack_empty. Inputs: stack. Output: empty stack. Purpose: sets the top of the stack to 0 so that the stack is empty.
- Function: stack_size. Inputs: stack. Output: the size of the stack (the top value). Purpose: calculates the size of the stack.
- Function: stack_copy. Inputs: destination pointer, source pointer. Outputs: top of both the destination and source stacks. Purpose: copies the values of the source array into the destination array.
- Function: stack_print. Inputs: stack, output file, cities array pointer. Output: city names. Purpose: prints the names of the cities from the graph.

path.c

- Function: path_create. Inputs: capacity. Outputs: path. Purpose: Creates the path by taking in the vertices and declares it as the length of the capacity of the stack. Calculates the total weight.
- Function: path_free. Inputs: pointer to the path array. Outputs: empty vertex pointer. Purpose: frees the pointer to a vertex.
- Function: path_vertices. Inputs: path pointer. Output: stack size. Purpose: returns the stack size after going through a vertex.

- Function: path_distance. Inputs: path pointer. Output: total weight. Purpose: returns the total weight of the path after going through a vertex.
- Function: path_add. Inputs: path pointer, value, graph pointer. Outputs: total weight/distance, start value, end value. Purpose: adds distance to the total weight and calculates it by calling graph_get_-weight and passing in the graph, start value, and end value parameters.
- Function: path_copy. Inputs: destination pointer, source pointer. Output: vertex name. Purpose: prints the vertices' names.
- Function: path_destroy. Inputs: pointer to array of path pointers. Outputs: empty path. Purpose: frees the memory allocated to the path array. Also frees the pointer.
- Function: path_remove. Inputs: path pointer, graph pointer. Output: removed vertex. Purpose: removes the vertex when the graph traverses backwards.

graph.c

- Function: graph_create. Inputs: number of vertices, directed boolean. Outputs: graph. Purpose: Creates the graph by allocating memory for the vertices, directed, visited, names, and weights. Stores the weights and returns the graph.
- Function: graph_free. Inputs: graph pointer. Outputs: freed graph and graph pointer. Purpose: frees the visited nodes, vertex names, weights, and graph. Also dereferences pointer.
- Function: graph_vertices. Inputs: graph. Outputs: graph vertices. Purpose: returns the graph vertices.
- Function: graph_add_vertex. Inputs: graph pointer, name pointer, value. Outputs: name of the vertex. Purpose: updates the name of the vertex after visiting it.
- Function: **graph_get_names. Inputs: graph pointer. Outputs: names list. Purpose: returns the list of vertex names.
- Function: graph_add_edge. Inputs: graph pointer, start value, end values, weight. Outputs: the weight of the edge. Purpose: uses the adjacency list to set the weight of the node by declaring it to the start and end indices of the weights array.
- Function: graph_get_weight. Inputs: graph pointer, start value, end value. Outputs:

tsp.c

- Function: main. Inputs: argc, *argc[]. Outputs: Shortest Hamiltonian cycle or error messages. Purpose: takes in the graph from the input and calls the functions to traverse the graph. If the input is invalid, prints the corresponding error messages. If input is valid, calls dfs function and prints the correct output.
- Function: *read_file. Inputs: filename, directed boolean. Outputs: vertex names, number of edges, weights, graph. Purpose: reads the input file, looks for the number of vertices, vertex names, number of edges, corresponding start, end, and weight values. Then, it prints them and closes the file. Returns the graph.
- Function: dfs. Inputs: graph pointer, start value, current value, path pointer, shortest path pointer, visited pointer. Outputs: shortest path. Purpose: goes through the graph by adding vertices into the path. Checks path distance and goes through each path recursively checking if it is the smallest path.

References